Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.437 Inference and Information
Spring 2017

Project: Cipher Breaking using Markov Chain Monte Carlo

Issued: Tuesday, April 25, 2017 **Part I Due:** Tuesday, May 2, 2017
Part II Due: Friday, May 12, 2017

- **Project Topic**
  This project explores the use of the Markov Chain Monte Carlo (MCMC) method to decrypt text encoded with a secret substitution cipher.

- **Document Structure**
  The introduction provides some background on substitution ciphers. Part I guides you through a Bayesian framework for deciphering a substitution cipher, and a basic MCMC-based approach to carrying it out. In Part II, you will further develop and refine your design from Part I.

- **Submission**
  Part I is to be submitted in class on the due date. Part II is to be submitted via Stellar by the specified due date. Be sure to follow submission guidelines in Part II *carefully*, since some key parts of the submission processing is automated.

- **Time Planning**
  With a total amount of time roughly equivalent to two problem sets, you should be able to investigate the key concepts, get some interesting results, have the opportunity to be creative, and have some fun! Part I is quite structured, but Part II is open-ended. For Part I, we strongly recommend you start working on it as early as possible so that you have enough time to fully debug your code and produce accurate results. For Part II, you should only go beyond that approximate amount of effort to the extent that you have the time and are motivated to explore in more detail.

- **Collaboration Policy**
  As with the homework more generally, you are permitted to discuss concepts and possible approaches with one or two of your classmates. However, you must code, evaluate, and write-up your final solutions individually.

- **Don't forget!**
  Completion of both parts of this project and the associated write-ups is a subject requirement.

- **Bonus**
  Special distinction will go to the best deciphering code submissions, measured in terms of accuracy and efficiency.

Introduction

A *substitution cipher* is a method of encryption by which units of *plaintext*—the original message—are replaced with *ciphertext*—the encrypted message—according to a *ciphering function*. The "units" may be single symbols, pairs of symbols, triplets of symbols, mixtures of the above, and so forth. The decoder deciphers the text by inverting the cipher.

For example, consider a simple substitution cipher that operates on single symbols. The ciphering function f is a one-to-one mapping between two finite and equal size alphabets $\mathcal{A}$ and $\mathcal{B}$. The plaintext is a string of symbols, which can be represented as a length-n vector $\mathbf{x} = (x_1, \ldots, x_n) \in \mathcal{A}^n$. Similarly, the ciphertext can be represented as vector $\mathbf{y} = (y_1, \ldots, y_n) \in \mathcal{B}^n$ such that

$$y_k = f(x_k), \qquad k = 1, 2, \ldots, n. \tag{1}$$

Without loss of generality, we assume identical alphabets, $\mathcal{A} = \mathcal{B}$, so that a ciphering function f is merely a permutation of symbols in $\mathcal{A}$. Throughout this project, we restrict our attention to the alphabet $\mathcal{A} = \mathcal{E} \cup \{\text{␣}, .\}$, where $\mathcal{E} = \{\text{a}, \text{b}, \ldots, \text{z}\}$ is the set of lower-case English letters. That is, our alphabet for the project is the collection of letters 'a' through 'z', space ' ', and period '.'. Henceforth any symbol to which we refer will be from this alphabet.

As an example, consider the short plaintext

this is a cool project.

This plaintext is encrypted using a substitution cipher into the ciphertext

.t qy qydywllpymnlsgwxb

where the ciphering function maps 't' into '.', 'h' into 't', 'i' into ' ', 's' into 'q', ' ' into 'y', '.' into 'b', and so on.

Deciphering a ciphertext is straightforward if the ciphering function f is known. Specifically, ciphertext $\mathbf{y} = (y_1, \ldots, y_n)$ is decoded as

$$x_k = f^{-1}(y_k), \qquad k = 1, 2, \ldots, n \tag{2}$$

where the deciphering or decoding function f^{-1} is the functional inverse of f. Note that this inverse exists because f is one-to-one. However, when f is a secret (i.e., unknown), decoding a ciphertext is more involved and is more naturally framed as a problem of *inference*. In this project, you will develop an efficient algorithm for decoding such secret ciphertexts.

Part I

Problem 1: Bayesian Framework

In this part we address the problem of inferring the plaintext x from the observed ciphertext $\mathbf{y}$ in a Bayesian framework. For this purpose, we model the ciphering function f as random and drawn from the uniform distribution over the set of permutations of the symbols in alphabet $\mathcal{A}$.

Assume that characters in English text can be approximately modeled as a simple Markov chain, so that the probability of a symbol in some text depends only on the symbol that precedes it in the text.

Enumerating the symbols in the alphabet $\mathcal{A}$ from 1 to $m = |\mathcal{A}|$ for convenience, the probability of transitioning from a symbol indexed with i to a symbol indexed with j is given by

$$\mathbb{P}\left(x_k = i \mid x_{k-1} = j\right) = M_{i,j}, \qquad i, j = 1, 2, \ldots, m, \qquad k \geq 2. \tag{3}$$

i.e., the ith row, jth column of the matrix $\mathbf{M} = [M_{i,j}]$ has the probability of transition from symbol j at time $k-1$ to symbol i at time k.

Moreover, assume that probability of the ith symbol in $\mathcal{A}$ is given by

$$\mathbb{P}\left(\mathsf{x}_k = i\right) = P_i, \qquad i = 1, 2, \ldots, m. \tag{4}$$

Finally, assume that $\mathbf{M}$ and $\mathbf{P} = [P_i]$ are known (given).

(a) Determine the likelihood of the (observed) ciphertext $\mathbf{y}$ under a ciphering function f, i.e., $p_{\mathsf{y}|\mathsf{f}}(\mathbf{y} \mid f)$.

(b) Determine the posterior distribution $p_{\mathsf{f}|\mathsf{y}}(f \mid \mathbf{y})$ and specify the MAP estimator $\hat{\mathsf{f}}_{\text{MAP}}(\mathbf{y})$ of the ciphering function f.

(c) Why is direct evaluation of the MAP estimate $\hat{\mathsf{f}}_{\text{MAP}}$ computationally infeasible?

Problem 2: Markov Chain Monte Carlo Method

Given the difficulty of directly evaluating the MAP estimate of the ciphering function, we use stochastic inference methods. As we learned in class, the MCMC framework is a convenient method for sampling from complicated distributions. This part guides you through the construction of a Metropolis-Hastings algorithm for the problem.

(a) Modeling the ciphering function f as uniformly distributed over the set of permutations of symbols from $\mathcal{A}$, find the probability that two ciphering functions f_1 and f_2 differ in exactly two symbol assignments.

(b) Using the Metropolis-Hastings algorithm, construct a Markov chain whose stationary distribution is the posterior found in Problem 1(b).
Hint: Use Problem 2(a) for constructing a proposal distribution.

(c) Fully specify the MCMC based decoding algorithm, in the form of pseudo-code.

Problem 3: Implementation

Implement the decoding algorithm developed in Problem 2 in `Matlab` or `Python`.

NOTE: The officially supported platform for the class project is `Matlab`. If you are comparatively inexperienced with implementing algorithms, we strongly recommend using this platform. However, for experienced users who prefer `Python` we offer this as an alternative platform, though we will not be able to offer the same level of support. Since we are offering the `Python` version as a pilot, if you choose this option, consider yourself an official beta-tester for us!

To help you test, debug, and refine your code, and to help you analyze and evaluate how the algorithm performs, we have provided the following data on Stellar in the file `project_part_I_matlab.zip`:
For Matlab:

- `language_parameters.mat`, which contains:
  - `alphabet`: a vector containing the alphabet symbols.
  - `letter_probabilities`: a vector containing probabilities of occurrence of alphabet symbols in a plaintext as defined in (4).
  - `letter_transition_matrix`: the transition probabilities $\mathbf{M} = [M_{i,j}]$ defined via (3).
- `example_cipher.mat`, which contains:
  - `plaintext`: an example plaintext.
  - `cipher_function`: the ciphering function f that was used to encrypt the plaintext.
  - `ciphertext`: the ciphertext obtained by applying `cipher_function` to `plaintext`.

For Python, the equivalent files in `project_part_I_python.zip` are

- `alphabet.csv`
- `letter_probabilities.csv`

- `letter_transition_matrix.csv`
- `cipher_function.csv`
- `ciphertext.txt`
- `plaintext.txt`

Each csv files has entries separated by a comma.

Run the algorithm on `ciphertext`. Use the plaintext and ciphering function, `cipher_function` provided to explore the convergence behavior of your algorithm.

(a) Plot the log-likelihood of the accepted state in MCMC algorithm versus iteration number.

(b) Plot the acceptance rate of state transitions in the MCMC algorithm versus iteration number. The acceptance rate $a(t)$ at iteration t is defined as the ratio between the number of accepted transitions between iterations $t - T$ and t and the overall number of proposed transitions, where window length T is appropriately chosen.

(c) Plot the accuracy rate versus iteration number. The accuracy rate $\beta(t)$ at iteration t is defined as the ratio between the number of correctly deciphered symbols with the ciphering function at iteration t and the overall length of the plaintext.

(d) Experiment with partitioning the ciphertext into segments and running your algorithm independently on different segments. Specifically, experiment with different segment lengths. How is the accuracy affected? Why?

(e) How does the log-likelihood per symbol, in bits, evolve over iterations? How does its steady-state value compare to the entropy of the English text? Explain. *Note:* The empirical entropy of English text depends on how the alphabet is modeled; for insights, see, e.g.,

> C. E. Shannon, "Prediction and Entropy of Printed English," *Bell System Technical Journal*, 1951.

Please make sure that you fully and clearly label all curves and axes in all your plots!

Note: You are not required to turn in your code for (this) Part I.

Part II

Building on the approach and insights developed in Part I, your goal in this part is to design an efficient decoder for deciphering a secret ciphertext, and implement your design in `Matlab` or `Python`. The ciphertext that your code will need to process comes from the application of a substitution cipher applied to English plaintext over the same 28-symbol plaintext and ciphertext alphabet of Part I.

As a reminder, this part is open-ended, giving you the opportunity to be thoughtful and creative in applying what you have learned.

Please carefully read the submission guidelines and how your code will be tested and evaluated, and ensure that your submission adheres to them strictly. Since the testing is automated, the evaluation of your submission will fail if the testing scripts cannot find your files, for instance!

While Developing Your Solutions...

- You may choose to make use of the transition probabilities $\mathbf{M}$ and marginal probabilities $\mathbf{P}$ from Part I if you like, but you are not required to. If you do not make use of them, be aware that all text we will encrypt to test your algorithm will be formatted so as to satisfy the following four constraints:
  - All text will be English.
  - The text always starts with a letter from the set $\mathcal{E}$.
  - A period ('.') is always followed by a space and always preceded by a letter.
  - A space is always followed by a letter.
- It is important for you to address how to stop your program; it should not iterate indefinitely. You need to develop a criterion for terminating your program when you are sufficiently confident of the decoded text. Problem 3(c) of Part I may, for example, provide some insight into this.
- A lot of information about the English language that may be useful to you can be found at `http://norvig.com/mayzner.html`. This has (among other things) n-gram tables (i.e., frequency of occurence of each possible combination of n letters at a time) for the English language, computed using a large repository. Of course, you do not have to use this information.
- To assist you in developing your decoder, three different "test" plaintexts are uploaded on Stellar in the file `project_part_II.zip`. You may generate your own ciphering functions, randomly or otherwise, to produce the corresponding ciphertexts, which you can subsequently use to test how your decoder performs

and refine it accordingly. Note that these are provided as a convenient resource in your development, and are *not* the plaintexts that we will use to evaluate your solutions!

Submission Guidelines

There are two items you must submit for (this) Part II.

(a) Submit your code in the form of a single file `upload.zip` via Stellar. Directly inside `upload.zip` you should have a `Matlab` function named `decode.m`, without any intermediate folders. In other words, when the command

```
unzip upload.zip -d ./test_directory
```

is executed at an Athena terminal, `test_directory` should contain the file `decode.m`. If you are using python, this file should be named `decode.py`.

To make such a zip file on Athena, you may use the command

```
zip upload.zip decode.m <other files to add>
```

The prototype of the function `decode` should be:

```
decode ( ciphertext , output_file_name )
```

where

- `ciphertext` is a ciphertext to be decoded.
- `output_file_name` is the name of a text file to which the output (i.e., the deciphered message) from your `decode.m` function should be written. The *full* file name will be provided; you should not modify it in any way!

  After your code runs, there should be a text file with name `output_file_name` in the current working directory. The text file should contain a single copy of the decoded text and nothing else. If you choose to output a deciphered plaintext intermittently, remember to overwrite what was written before.

Note that in python, your `decode.py` must contain a function defintion of the form `def decode(ciphertext, output_file_name)`, where the arguments are the same as above. When writing, you should use

```
f = open(output_file_name, 'w')
f.write(<your deciphered message>)
f.close()
```

Make sure that your submission contains all the files needed by the decoder, including any files from Part I that might be required by your decoder. And remember to use *relative* pathnames so that your `decode` function can find any other functions that it needs!

Additionally, be sure to *remove all functions related to figures* in your submitted code, such as `figure` and `plot`. These functions are not supported in the automated evaluation platform and thus will lead to a crash that prevents your code from being evaluated.

(b) Submit a short (approximately 2-3 page) description of your methodology and algorithm. This description should include details of what refinements and enhancements you implemented to improve performance of the basic decoder, and a discussion of how you analyzed, evaluated, optimized, and tested it. This must also be turned in by the submission deadline.

Code Testing Guidelines

To confirm that your code can be evaluated by our automated platform, we **strongly recommend** that you test your code following the steps (a)-(c) below before making your submission via Stellar. You should be able to perform this test from your own computer. However, it should be emphasized that this test only detects problems with your code that will lead to the system failing to evaluate your submission; thus, passing this test does not guarantee the correctness of the deciphered result.

(a) **Prepare:**

- Download the file `test_code_part_II.zip` from Stellar. After unzipping it, you should find a folder `test_code_part_II` that has three files: `test_code.sh`, `matlab_wrapper.m`, and `coded_message.mat`.
- Copy the file `upload.zip` that you will submit on Stellar to the folder `test_code_part_II`. Do *not* use another name for your `.zip` file.

(b) **Copy to Athena:**
Copy the folder `test_code_part_II` with the four files in it onto Athena. If you don't already have your favorite way of doing this, possible approaches for this step include using file transfer tools such as `SecureFX` for Windows and `Fetch` for Mac, which can be downloaded from

- 64-bit Windows: `http://ist.mit.edu/securecrt-fx/win64/recommended`
- 32-bit Windows: `http://ist.mit.edu/securecrt-fx/win32/recommended`
- MacOS X: `http://ist.mit.edu/fetch/5x/mac`

These tools from the IST website have been pre-configured with a session profile with which you can directly login to your home directory on Athena. Run these tools, login with your Kerberos account, and then you could put the folder `test_code_part_II` in your home directory on Athena.

Alternatively, you can access Athena enviroment using any Athena workstations. There are a lot of them in the campus.

(c) **Test Code on Athena:**

- Visit `https://athena.dialup.mit.edu/` in your web browser. A terminal will appear on the page, and then login with your Kerberos account.
- In the terminal, go to your folder `test_code_part_II` on Athena that you copied in step (b). For example, if this folder is put directly under your home directory, then you can execute the command below

```
cd ~/test_code_part_II
```

- Execute the command below in the terminal

```
source test_code.sh
```

After you see "`Academic License`" in the terminal, wait for your code to finish. If it is the first time that you run `Matlab` on Athena, then you may need to press any key to skip some introduction information and see "`Academic License`". If it finally displays "`Program terminates normally!`" in the terminal, then your code has passed this basic compatability test. However, if there are error messages at the end (starting from a line with "`--- ERROR! ---`"), then there should be either problems in your matlab scripts that cause `Matlab` to crash, or required files that are missing from the path. **As a minimal check for compatibility, be sure that your code passes this test before submission!**

Code Evaluation

Each submitted decoder will be run on `MATLAB_R2016b` (version 9.1) or `Python 2.7.6` on an Athena and applied to a collection of different ciphertexts we create from different plaintexts and ciphers.

For reference, a typical, well-engineered decoder you might submit should converge accurately within at most a few minutes. Nevertheless, we will allow each student submission up to 1 hour of run time, as to avoid unduly penalizing inefficient but otherwise good implementations. This is the time allotted per student to start `Matlab` and call the function for *all* of the ciphertexts, one at a time. Hence, if your decoder doesn't have a built in stopping criterion, it will terminate after an hour, and the deciphered text in the output file (or files, if some ciphertexts have finished) at that point will be used. NOTE: please do *not* estimate the run time of your code in the evaluation from its execution time on the Athena dialup server (`https://athena.dialup.mit.edu/`) used in code testing, since a different and more powerful server will be used for evaluation.

The criteria for the evaluation emphasize the accuracy of the deciphered results. We will measure how accurately your coded algorithm can decode each of the different ciphertexts we provide as input. More specifically, we will measure the accuracy rate for each ciphertext, defined as the ratio between the number of correctly deciphered symbols and the overall number of symbols in a plaintext. In addition to accuracy, our evaluation will also consider the computational running time of your solution—in particular, the time your method requires to process each ciphertext.

Finally...

Have fun! We will highlight some of the best and most interesting submitted solutions in class!